AIL861/ELL8299/ELL881: Advances in Large Language Models

Mid-Semester Examination: Solutions

Semester I, 2025–26, IIT Delhi

Section-A

Multiple Choice Questions (6 marks)

Each question carries 1.5 marks. MORE THAN ONE OPTION CAN BE CORRECT FOR EACH QUESTION. There is NO PARTIAL MARKING – only marking ALL options correctly will fetch marks.

1. Consider a Seq2Seq model with attention. The encoder hidden states for 3 tokens are $s_1 = [2, -1, 0], s_2 = [-1, 2, 1], s_3 = [1, 1, -1]$. The attention weights at decoder timestep t are $\alpha = [0.4, 0.3, 0.3]$.

   Which of the following statement(s) is/are correct with respect to Context vector c_t?

   (A) $c_t = [0.8, 0.5, 0]$
   (B) c_t cannot be computed because the decoder hidden state is not given.
   (C) In a Seq2Seq model, scaling the encoder vectors by a constant factor does not change the resulting c_t direction.
   (D) Using attention mechanism in Seq2Seq models helps to counter the bottleneck problem.

   Answer

   Correct option(s): (A), (D)
   Justification:
   (A) Correct — $c_t = \sum_i \alpha_i s_i = 0.4[2, -1, 0] + 0.3[-1, 2, 1] + 0.3[1, 1, -1] = [0.8,\ 0.5,\ 0]$.
   (B) Incorrect — Since α are given, the decoder hidden state is not needed to compute c_t.
   (C) Incorrect — in a Seq2Seq model, attention weights α come from a softmax over similarity scores (e.g., $q_t^\top s_i$). Scaling s_i rescales these logits, which generally changes the softmax distribution α and thus can change the direction of c_t.
   (D) Correct — attention lets the decoder access all encoder states directly, mitigating the encoder bottleneck.

2. Which of the following statement(s) regarding tokenization strategies is/are correct?

   (A) Subword tokenization methods like BPE, WordPiece, and unigram reduce out-of-vocabulary (OOV) issues compared to word-level tokenization.
   (B) In Byte-Pair Encoding (BPE), we only store the final vocabulary and discard the merge rules.
   (C) In the unigram subword model, tokens are added iteratively until the vocabulary reaches the desired size.
   (D) In WordPiece, if a word contains any subword not in the vocabulary, the entire word is replaced with an `<UNK>` token.

Answer

Correct option(s): (A), (D)
Justification: (A) Correct — Subword methods (BPE, WordPiece, Unigram) reduce OOV problems compared to pure word-level tokenization.
(B) Incorrect — BPE requires storing the sequence of merge rules to reconstruct tokenization, not just the final vocabulary.
(C) Incorrect — the unigram model starts with a large candidate vocab and prunes tokens, rather than adding them iteratively.
(D) Correct — in WordPiece, if a word cannot be decomposed into known subwords, it is replaced with a single `<UNK>` token.

3. You have a transformer with attention weight matrices of size 120×120. You want to fine-tune these matrices using LoRA, in such a way that the number of trainable parameters is reduced by a factor of 15. Given, r denotes the rank of the low-rank decomposition, Which of the following statement(s) is/are correct?

(A) Setting $r = 2$ achieves the required reduction in learnable parameters.
(B) Setting $r = 4$ achieves the required reduction in learnable parameters.
(C) During inference, old weights are discarded and replaced by LoRA weights.
(D) During inference, the old weights are kept and the LoRA updates are added to them.

Answer

Correct option(s): (B), (D)
Justification:
(A) Incorrect — Full matrix has $120 \times 120 = 14400$ params. LoRA adds $120r + r120 = 240r$ trainable params. A $15\times$ reduction means $14400/(240r) = 15 \Rightarrow r = 4$, not 2.
(B) Correct — Shown above.
(C) Incorrect — At inference we do *not* discard the base weights.
(D) Correct — Inference uses $W_{\text{eff}} = W + \Delta W$ with ΔW from the LoRA low-rank factors (added to the frozen base W).

4. Consider the RLHF objective

$$J(\pi_\theta) = \mathbb{E}_{y\sim\cdot}\left[r(x, y) - \beta \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}\right].$$

Which of the following statement(s) is/are correct?

(A) The exact value of $\nabla_\theta J(\pi_\theta)$ cannot be computed in closed form because it involves an expectation over all possible sequence.
(B) The exact value of $\nabla_\theta J(\pi_\theta)$ can be computed in closed form using the log-derivative trick.
(C) Increasing the scaling factor β to a very large value can cause the model to exploit or hack the reward model.
(D) The expectation is taken over the reference policy π_{ref}
(E) The expectation is taken over the base policy π_θ

Answer

Correct option(s): (A), (E)
Justification:
(A) Correct — The gradient involves an expectation over all possible output sequences; exact closed-form evaluation is infeasible, so we use sampling-based estimators.
(B) Incorrect — The log-derivative (REINFORCE) trick yields an *unbiased estimator* of the gradient, not a closed-form value.
(C) Incorrect — A very large β strongly regularizes toward π_{ref}, thus forcing it to stay close to the reference LLM.
(D) Incorrect — The expectation is *not* over π_{ref}.
(E) Correct — The expectation is taken over the current (base) policy $\pi_\theta(y|x)$.

Section-B

(39 marks)

Question 1: Recurrent Neural Networks (3 marks)

(a) Consider a recurrent neural network (RNN) defined by

$$h_t = Ah_{t-1} + Bx_t + b, \qquad y_t = Ch_t,$$

where $x_t \in \mathbb{R}^d$ is the input at time step t, and $\hat{y}_t$ denotes the ground-truth target corresponding to y_t. The sequence length (number of tokens) is T. The loss function is given by

$$L = \frac{1}{2}\sum_{t=1}^{T} \|y_t - \hat{y}_t\|^2.$$

Derive an expression for $\frac{\partial L}{\partial A}$. Your steps should be clearly shown, and you may introduce and use intermediate gradient expressions where appropriate. *(3 marks)*

Answer

Step 1: Gradient wrt outputs

$$\boxed{\frac{\partial L}{\partial y_t} = y_t - \hat{y}_t} \qquad (t = 1, \dots, T)$$

Step 2: Backprop through $y_t = Ch_t$ and the recurrence $h_{t+1} = Ah_t + Bx_{t+1} + b$

$$\boxed{\frac{\partial L}{\partial h_t} = C^\top \frac{\partial L}{\partial y_t} + \frac{\partial L}{\partial h_{t+1}}\frac{\partial h_{t+1}}{\partial h_t} = C^\top (y_t - \hat{y}_t) \, + \, A^\top \frac{\partial L}{\partial h_{t+1}}}$$

Step 3: Collect terms for $\frac{\partial L}{\partial A}$

$$\boxed{\frac{\partial L}{\partial A} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial A} = \sum_{t=1}^{T} \Big(\frac{\partial L}{\partial h_t}\Big)\, h_{t-1}^\top}$$

Question 2: Transformer Architecture (10 marks)

(a) With respect to Positional encoding in Transformers,

(i) Explain the role of positional encoding in the Transformer architecture and why it is necessary.

(ii) A simple approach to positional encoding is to associate each token position with an integer value (i.e., $1, 2, 3, \dots$). Discuss a limitation of this scheme. *(1+1 = 2 marks)*

Answer

(i) Positional encoding is necessary because self-attention alone is permutation-invariant; it gives the model a notion of token order.

(ii) Using raw integers (1,2,3,...) is limited since it does not generalize well to longer/variable-length sequences, and large position values can bias the model toward later tokens.

(b) Explain the purpose of the Add & Norm block within a Transformer layer. In addition, discuss why layer normalization is typically used in Transformer models instead of batch normalization. *(2 marks)*

Answer

The Add & Norm block combines residual connections with layer normalization. Residuals help stabilize training and preserve gradients, while normalization keeps activations well-scaled.

Layer normalization is preferred over batch normalization because it normalizes across features of each token independently, making it effective for variable-length sequences and small batch sizes where batch statistics would be unreliable.

(c) In the scaled dot-product attention mechanism, the attention score between a query vector $Q \in \mathbb{R}^{d_k}$ and a key vector $K \in \mathbb{R}^{d_k}$ is computed as

$$\text{score}(Q, K) = \frac{QK^\top}{\sqrt{d_k}}.$$

Assume that the entries of Q and K are independent random variables with zero mean and unit variance. Show, with proper steps, that the variance of score(Q, K) is also 1. Also, explain why the scaling by $\frac{1}{\sqrt{d_k}}$ is necessary. *(3 marks)*

Answer

Let $\mathrm{S} := \mathrm{QK}^\top = \sum_{i=1}^{d_k} Q_i K_i$, and $\text{score}(Q, K) = \frac{S}{\sqrt{d_k}}$.
Variance of S. Let $Z_i := Q_i K_i$. Each pair (Q_i, K_i) is independent of (Q_j, K_j) for $i \neq j$, so the Z_i are independent. Also,

$$\mathrm{Var}(Z_i) = \mathbb{E}[Q_i^2 K_i^2] - (\mathbb{E}[Q_i K_i])^2 = \mathbb{E}[Q_i^2] \cdot \mathbb{E}[K_i^2] = 1 \cdot 1 = 1.$$

Thus,

$$\mathrm{Var}(S) = \sum_{i=1}^{d_k} \mathrm{Var}(Z_i) = d_k.$$

Variance of the scaled score.

$$\mathrm{Var}\left(\frac{S}{\sqrt{d_k}}\right) = \frac{1}{d_k}\mathrm{Var}(S) = \frac{1}{d_k} \cdot d_k = 1.$$

Why the $1/\sqrt{d_k}$ scaling? Without scaling, $\mathrm{Var}(QK^\top) = d_k$ grows with head dimension, causing attention logits to have large magnitude, which pushes softmax toward saturation and yields poor/unstable gradients. Dividing by $\sqrt{d_k}$ keeps the logit variance ≈ 1, stabilizing softmax and gradients across different d_k.

(d) Consider a single-head Transformer layer with input $X \in \mathbb{R}^{n\times d}$ and projection matrices $W_Q, W_K, W_V \in \mathbb{R}^{d\times d}$ for computing queries, keys, and values. The layer also contains a feedforward network (FFN) with hidden dimension $d_{ff} = 4d$.

(i) Derive with clear steps, the total time complexity of executing this single head Transformer layer.

(ii) Suppose the Transformer now uses h attention heads instead of a single head, where each head has its own projection matrices $W_Q^{(i)}, W_K^{(i)}, W_V^{(i)} \in \mathbb{R}^{d\times d'}$, with $d' = d/h$. Determine the new time complexity in this case. *(1.5 + 1.5 = 3 marks)*

Answer

(i) Single head (sequence length n, model dim d, FFN hidden $4d$):

- Project Q,K,V: $XW_Q, XW_K, XW_V \to 3\,O(nd^2)$
- Attention scores: $QK^\top \to O(n^2d)$
- Apply attention to values: $\text{Attn}\cdot V \to O(n^2d)$
- FFN ($d \to 4d \to d$): $O(nd\cdot 4d) + O(n\cdot 4d\cdot d) = O(8nd^2)$

Total: $O(n^2d) + O(nd^2)$

(ii) Multi-head (h heads, per-head dim $d' = d/h$):

- Project all heads (Q,K,V): $3h\cdot O(ndd') = 3\,O(nd^2)$
- Attention scores (all heads): $h\cdot O(n^2d') = O(n^2d)$
- Apply attention to values (all heads): $h\cdot O(n^2d') = O(n^2d)$
- FFN ($d \to 4d \to d$): $O(8nd^2)$

Total: $O(n^2d) + O(nd^2)$

Question 3: Model Tuning (3 marks)

(a) Suppose you want to use the pre-trained BERT for a sentiment classification task using the IMDb movie reviews dataset. Explain how you would fine-tune BERT for this task. *(1 mark)*

Answer

Use the pre-trained BERT model and take the final hidden state of the [CLS] token as the sequence representation. Add a linear layer on top of this [CLS] embedding to predict sentiment (positive/negative) using softmax. Fine-tune both BERT and the added linear layer on the IMDb dataset with cross-entropy loss.

(b) Explain the entire Self-Instruct process for generating instruction-following data. *(2 marks)*

Answer

Self-Instruct begins with a small set of seed instructions. 1. The LM is prompted to generate new diverse instructions from these seeds. 2. Each new instruction is categorized (classification vs. non-classification). 3. The LM then generates corresponding input–output examples (via input-first or output-first prompting). 4. Low-quality or duplicate samples are filtered out. 5. The final curated data is used to fine-tune the LM, improving its ability to follow unseen instructions.

Question 4: Aligning LLMs with Human Preferences (10 marks)

(a) Assume a reward model assigns a real-valued score $r_\theta(x, y)$ to each response y for a given input x. You are given a dataset $\mathcal{D}$ of ranked responses (x, y_1, y_2, y_3), where the responses are ranked as $y_1 > y_2 > y_3$. Using the Bradley-Terry model, write the log-likelihood that maximizes the probability of the preferred responses . *(2 marks)*

Answer

Bradley–Terry pairwise prob: $\Pr(y_i \succ y_j \mid x) = \dfrac{e^{r_\theta(x,y_i)}}{e^{r_\theta(x,y_i)} + e^{r_\theta(x,y_j)}} = \sigma\big(r_\theta(x, y_i) - r_\theta(x, y_j)\big).$

For a ranked triple $y_1 > y_2 > y_3$: use independent pairwise prefs $(y_1 \succ y_2),\ (y_1 \succ y_3),\ (y_2 \succ y_3)$.

$$\mathcal{L}(\theta) = \prod_{(x,y_1,y_2,y_3)\in\mathcal{D}} \underbrace{\Pr(y_1 \succ y_2 \mid x)}_{=\sigma(r_\theta(x,y_1)-r_\theta(x,y_2))} \underbrace{\Pr(y_1 \succ y_3 \mid x)}_{=\sigma(r_\theta(x,y_1)-r_\theta(x,y_3))} \underbrace{\Pr(y_2 \succ y_3 \mid x)}_{=\sigma(r_\theta(x,y_2)-r_\theta(x,y_3))} .$$

Log-likelihood:

$$\log \mathcal{L}(\theta) = \sum_{(x,y_1,y_2,y_3)\in\mathcal{D}} \Big[\log \sigma\big(r_\theta(x, y_1) - r_\theta(x, y_2)\big) + \log \sigma\big(r_\theta(x, y_1) - r_\theta(x, y_3)\big)$$
$$+ \log \sigma\big(r_\theta(x, y_2) - r_\theta(x, y_3)\big)\Big].$$

(b) Explain the need of importance weights in policy gradient methods. Write the modified expression for the REINFORCE gradient estimation using importance sampling weights. *(2 marks)*

Answer

In LLM training with policy gradients, we often generate rollouts from a reference (behavior) policy π_{ref} instead of the current policy π_θ (to reuse data or reduce cost). Since the data distribution differs, importance sampling weights

$$w_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\text{ref}}(a_t|s_t)}$$

are applied to correct for this mismatch, ensuring the gradient estimate is unbiased.
The REINFORCE gradient with importance sampling for a trajectory is:

$$\nabla_\theta J(\pi_\theta) \;=\; \mathbb{E}_{\tau\sim\pi_{\text{ref}}}\left[\sum_{t=0}^{T} \underbrace{\frac{\pi_\theta(a_t|s_t)}{\pi_{\text{ref}}(a_t|s_t)}}_{w_t} \hat{A}(s_t, a_t)\, \nabla_\theta \log \pi_\theta(a_t|s_t)\right].$$

Here $\hat{A}(s_t, a_t)$ is the estimated advantage (e.g. from rewards in RLHF rollouts).

(c) Consider a parameterized value function network $V_\phi(s)$ that estimates the expected return from each state s. You observe a single trajectory $(s_0, a_0) \to (s_1, a_1) \to (s_2, a_2) \to s_3$ (terminal). The rewards are $r(s_0, a_0) = 3,\ r(s_1, a_1) = -1,\ r(s_2, a_2) = 2$, and the discount factor is $\gamma = 0.95$. The current value network outputs $V_\phi(s_0) = 4.0,\ V_\phi(s_1) = 1.0,\ V_\phi(s_2) = 0.5,\ V_\phi(s_3) = 0$.

(i) Calculate the advantage $A(s_0, a_0)$ using the full Monte Carlo estimate.

(ii) Calculate the advantage $A(s_0, a_0)$ using the 2-step TD estimate. *(1.5 + 1.5 = 3 marks)*

Answer

Given: $\gamma = 0.95,\ r_0 = 3,\ r_1 = -1,\ r_2 = 2,\ V_\phi(s_0) = 4.0,\ V_\phi(s_1) = 1.0,\ V_\phi(s_2) = 0.5,\ V_\phi(s_3) = 0.$

(i) **Full Monte Carlo:**

$$G_0 = r_0 + \gamma r_1 + \gamma^2 r_2 = 3 + 0.95(-1) + 0.95^2(2) = 3 - 0.95 + 1.805 = 3.855.$$

$$A(s_0, a_0) = G_0 - V_\phi(s_0) = 3.855 - 4.0 = \boxed{-0.145}.$$

(ii) **2-step TD:**

$$G_0^{(2)} = r_0 + \gamma r_1 + \gamma^2 V_\phi(s_2) = 3 + 0.95(-1) + 0.95^2(0.5) = 3 - 0.95 + 0.45125 = 2.50125.$$

$$A(s_0, a_0) = G_0^{(2)} - V_\phi(s_0) = 2.50125 - 4.0 = \boxed{-1.49875}\ (\approx -1.5).$$

(d) In Direct Preference Optimization (DPO), the intermediate reward is removed, and learning is performed directly from preference comparisons. Let π_θ denote the current policy, π_{ref} the reference policy, and $\mathcal{D} = \{(x, y^+, y^-)\}$ a dataset of preference pairs, where y^+ is preferred over y^-.

(i) Using Langragian, express the optimal reward model as a function of the optimal policy.

(ii) Using the expression for optimal reward, derive the training objective for DPO.
(1.5 + 1.5 = 3 marks)

Answer

Start from the KL–regularized policy objective for a fixed prompt x:

$$\pi^*(\cdot|x) \;=\; \arg\max_{\pi(\cdot|x)} \mathbb{E}_{y\sim\pi(\cdot|x)}\big[r^*(x,y)\big] \;-\; \beta\,\mathrm{KL}\big(\pi(\cdot|x)\,\|\,\pi_{\text{ref}}(\cdot|x)\big) \quad \text{s.t.} \quad \sum_y \pi(y|x) = 1.$$

Lagrangian:

$$\mathcal{L}(\pi,\lambda) = \sum_y \pi(y|x)\,r^*(x,y) - \beta\sum_y \pi(y|x)\Big(\log\pi(y|x) - \log\pi_{\text{ref}}(y|x)\Big) + \lambda\left(\sum_y \pi(y|x) - 1\right).$$

(i) Optimal reward in terms of the optimal policy. Set $\partial\mathcal{L}/\partial\pi(y|x) = 0$ at $\pi = \pi^*$:

$$r^*(x,y) \;-\; \beta\Big(1 + \log\tfrac{\pi^*(y|x)}{\pi_{\text{ref}}(y|x)}\Big) \;+\; \lambda \;=\; 0 \;\Rightarrow\; \boxed{r^*(x,y) = \beta\log\frac{\pi^*(y|x)}{\pi_{\text{ref}}(y|x)} + (\beta - \lambda)}.$$

(ii) DPO objective (preference pairs (x, y^+, y^-)). Using the Bradley–Terry model,

$$\Pr(y^+ \succ y^- \mid x) = \sigma\Big(\tfrac{1}{\beta}\big(r_\theta(x,y^+) - r_\theta(x,y^-)\big)\Big).$$

Substitute the implicit reward $r_\theta(x,y) = \beta\log\frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)} + \beta\log Z_x(\theta)$; the x–only $\beta\log Z_x(\theta)$ cancels:

$$\Pr(y^+ \succ y^- \mid x) = \sigma\left(\log\frac{\pi_\theta(y^+|x)}{\pi_{\text{ref}}(y^+|x)} - \log\frac{\pi_\theta(y^-|x)}{\pi_{\text{ref}}(y^-|x)}\right).$$

Hence the DPO training objective is

$$\boxed{\max_\theta \frac{1}{|\mathcal{D}|}\sum_{(x,y^+,y^-)\in\mathcal{D}} \log\sigma\Big(\log\tfrac{\pi_\theta(y^+|x)}{\pi_{\text{ref}}(y^+|x)} - \log\tfrac{\pi_\theta(y^-|x)}{\pi_{\text{ref}}(y^-|x)}\Big)}.$$

Question 5: Efficient LLMs (10 marks)

(a) Consider a **decoder-only Transformer** language model that generates text token-by-token.

Notations

- L: number of Transformer layers
- H: number of attention heads
- d_{model}: model width/dimension
- d_k: per-head key/query dimension
- V: Vocabulary size
- $W_{\text{out}} \in \mathbb{R}^{d_{\text{model}} \times V}$: output/unembedding projection
- For layer ℓ and attention head h: projections $W_Q^{\ell(h)}, W_K^{\ell(h)}, W_V^{\ell(h)}, W_O^{\ell}$
- Input prompt tokens: $\mathbf{y}_{1:m}$; generation proceeds for $t = m+1, m+2, \ldots$ up to $T_{\max}$ or until an `<eos>` token is produced.
- Token selection: use greedy decoding, i.e., $\arg\max$ over the softmax logits.

(i) Write clean `pseudocode` for next-token generation in the following two settings:

(1) *Without KV caching* *(2 marks)*
At each decoding step, run a full forward pass over the *entire* prefix to produce the next token. Provide pseudocode that clearly shows the embedding+positional encoding, per-layer masked self-attention over all positions seen so far, feed-forward/residual/LN structure, logits computation, greedy selection, and stopping criterion.

Answer

```
function GENERATE_NO_CACHE(y_1:m, Tmax):
    # Embed + add positions for the whole (growing) prefix every step
    t = m + 1
    Y = y_1:m
    while t <= Tmax:
        X = Embed(Y) + PosEnc(len(Y))      # shape: [len(Y), d_model]
        # L Transformer blocks with masked self-attention
        for l in 1..L:
            # compute Q,K,V for all positions 1..len(Y)
            # each: [len(Y), H, d_k]
            Q = X @ W_Q[l];  K = X @ W_K[l];  V = X @ W_V[l]
            # causal mask: positions attend to <= their index
            for i in 1..len(Y):
                # scores over keys 1..i for each head
                scores[i, 1..i, h] = (Q[i,h] · K[1..i,h]^T) / sqrt(d_k)
                attn[i,h] = softmax(scores[i,1..i,h])   # [i]
                Z[i,h] = attn[i,h]^T @ V[1..i,h]   # [d_k]
            Z = ConcatHeads(Z) @ W_O[l]            # [len(Y), d_model]
            X = LN( X + Z )                        # Add & Norm
            # Position-wise FFN
            F = FFN(X)                             # [len(Y), d_model]
            X = LN( X + F )                        # Add & Norm
        # logits at last position only
        x_last = X[len(Y)]
        logits = x_last @ W_out                    # [V]
        y_t = argmax(softmax(logits))
        append Y <- y_t
        if y_t == <eos>: break
        t = t + 1
    return Y
```

(2) *With KV caching* *(4 marks)*

Maintain per-layer caches of past keys/values so that at step t you compute only the current query q_t and attend to cached $K_{1:t}$ and $V_{1:t}$. Provide pseudocode that shows cache initialization (including warming with the prompt), cache usage at each step, cache updates, logits computation, greedy selection, and stopping criterion.

Answer

```
function GENERATE_WITH_CACHE(y_1:m, Tmax):
    # For each l and head h, store K_cache[l][h], V_cache[l][h]
    init empty K_cache, V_cache
    # -- Warm the cache with the prompt (one forward over positions 1..m)
    X = Embed(y_1:m) + PosEnc(m)                              # [m, d_model]
    for l in 1..L:
        Qp = X @ W_Q[l]; Kp = X @ W_K[l]; Vp = X @ W_V[l]  # [m, H, d_k]
        # causal attention over prompt to form prompt hidden states
        # (optional if we only need K,V)
        # Update caches with full prompt keys/values
        K_cache[l] = Kp                                         # [m, H, d_k]
        V_cache[l] = Vp                                         # [m, H, d_k]
        # Compute Z over prompt (causal) to produce X for next layer
        for i in 1..m:
            scores = (Qp[i] · Kp[1..i]^T) / sqrt(d_k)
            attn   = softmax(scores)
            Z[i]   = (attn @ Vp[1..i])
        X = LN( X + (ConcatHeads(Z) @ W_O[l]) )
        X = LN( X + FFN(X) )
    Y = y_1:m
    t = m + 1
    # -------------------- Autoregressive decoding ---------------------
    while t <= Tmax:
        # Embed current token position only (we only need x_t)
        x_t = Embed(Y[t-1]) + PosEnc(t)                         # [d_model]
        for l in 1..L:
            q_t = x_t @ W_Q[l]                                  # [H, d_k]
            k_t = x_t @ W_K[l]                                  # [H, d_k]
            v_t = x_t @ W_V[l]                                  # [H, d_k]
            # Append to caches (no recomputation of earlier keys/values)
            K_cache[l].append(k_t)                         # now length = t
            V_cache[l].append(v_t)
            # Attend only at position t to all cached keys/values
            scores = (q_t · K_cache[l]^T) / sqrt(d_k)         # [H, t]
            attn   = softmax(scores)                            # [H, t]
            z_t_heads = attn @ V_cache[l]                       # [H, d_k]
            z_t = ConcatHeads(z_t_heads) @ W_O[l]               # [d_model]
            x_t = LN( x_t + z_t )
            f_t = FFN(x_t)
            x_t = LN( x_t + f_t )
        logits = x_t @ W_out                                    # [V]
        y_t = argmax(softmax(logits))                           # greedy
        append Y <- y_t
        if y_t == <eos>: break
        t = t + 1
    return Y
```

(ii) *Complexity and memory comparison* *(2 marks)*

State and briefly justify the asymptotic *time* complexity as a function of target length T (beyond the prompt) for both settings, and the *additional memory* required to store KV caches. Use big-O notation and specify the main parameters involved (e.g., L, H, d_k, etc.).

Answer

Let T be the number of tokens to be generated, m be the length of the input prompt, L be the number of layers, H be the number of attention heads, and d_k be the dimension of the key/query vectors for each head. The total model dimension is $d_{\text{model}} = H \cdot d_k$.

Inference Without KV Caching

In this setting, for each new token generated, the entire sequence (prompt + previously generated tokens) is re-processed from scratch.

Time Complexity: At each step t (for $t = 1, \ldots, T$), the model processes a sequence of length $k = m + t$. The self-attention mechanism, which dominates the computation, has a complexity quadratic in the sequence length, i.e., $O(k^2 \cdot d_{\text{model}})$.

The total time complexity is the sum of the costs for generating each of the T tokens:

$$\text{Total Time} \propto \sum_{t=1}^{T} \text{Cost(step } t) = \sum_{t=1}^{T} O((m+t)^2 \cdot d_{\text{model}})$$

Ignoring the constant prompt length m for asymptotic analysis with respect to T, this simplifies to:

$$\text{Total Time} \propto d_{\text{model}} \sum_{t=1}^{T} t^2 \approx d_{\text{model}} \cdot \frac{T^3}{3}$$

Thus, the overall time complexity is cubic in the number of generated tokens:

$$\text{Time Complexity} = O(L \cdot d_{\text{model}} \cdot T^3)$$

Additional Memory Complexity: Since no intermediate states (Keys and Values) are stored between generation steps, the additional memory required is constant.

$$\text{Additional Memory} = O(1)$$

Inference With KV Caching

With KV caching, the Key and Value vectors for all past tokens are stored and reused. At each step, we only need to compute the Q, K, and V vectors for the single newest token.

Time Complexity: At each step t (for $t = 1, \ldots, T$), we compute the query vector for the new token and attend to the cached Keys and Values from all $k - 1 = m + t - 1$ previous tokens. This attention operation is linear in the sequence length, with a cost of $O(k \cdot d_{\text{model}})$.

The total time complexity is the sum of these linear costs:

$$\text{Total Time} \propto \sum_{t=1}^{T} \text{Cost(step } t) = \sum_{t=1}^{T} O((m+t) \cdot d_{\text{model}})$$

Ignoring the constant prompt length m, this simplifies to:

$$\text{Total Time} \propto d_{\text{model}} \sum_{t=1}^{T} t \approx d_{\text{model}} \cdot \frac{T^2}{2}$$

Thus, the overall time complexity is quadratic in the number of generated tokens:

$$\text{Time Complexity} = O(L \cdot d_{\text{model}} \cdot T^2)$$

Additional Memory Complexity: The memory is required to store the Key and Value tensors for every token in the sequence, for each layer and each head. After generating T tokens, the total sequence length is $m + T$.

$$\text{KV Cache Memory} = L \times H \times (m+T) \times d_k \times (\text{bytes per element})$$

Answer

The additional memory complexity is therefore:

$$\text{Additional Memory} = O(L \cdot H \cdot T \cdot d_k) = O(L \cdot T \cdot d_{\text{model}})$$

(b) Contrast stage-1, stage-2 and stage-3 of ZeRO (Zero Redundancy Optimizer) based on the – (i) sharded parameters, (ii) memory requirements, and (iii) communication cost. Assume P to be the total number of model parameters and N to be the number of available GPUs. Also assume that we are doing mixed-precision training using `bfloat16` for computations and `float32` for storage. *(2 marks)*

Answer

Let P be the total parameters and N GPUs. Mixed precision: compute in `bf16`, maintain optimizer states and master weights in `fp32`.

ZeRO stage	**Sharded components**	**Memory Requirements (excluding activations)**	**Communication Cost**
Stage-1	Batch & Optimizer states	$2P + 2P + \frac{12P}{N}$	$\approx 2P$
Stage-2	Batch & Optimizer states & gradients	$2P + \frac{2P+12P}{N}$	$\approx 2P$
Stage-3	Batch & Optimizer states & gradients & model parameters	$\frac{2P+2P+12P}{N}$	$\approx 3P$

Question 6: PEFT Techniques and Compression (3 marks)

(a) What is the difference between Prompt-Tuning and Prefix-Tuning? *(1 mark)*

Answer

Prompt-tuning learns a small set of continuous prompt embeddings prepended to the input, while prefix-tuning learns trainable prefix vectors that modify the key/value states at every Transformer layer. Prompt-tuning only adjusts embeddings at the input layer, whereas prefix-tuning influences the model throughout its depth.

(b) You have designed an algorithm that takes a transformer model and an integer p as inputs and performs unstructured pruning by masking p% of the model weights based on their magnitude. This algorithm is applied to all weight tensors in the model. Based on your assumptions:

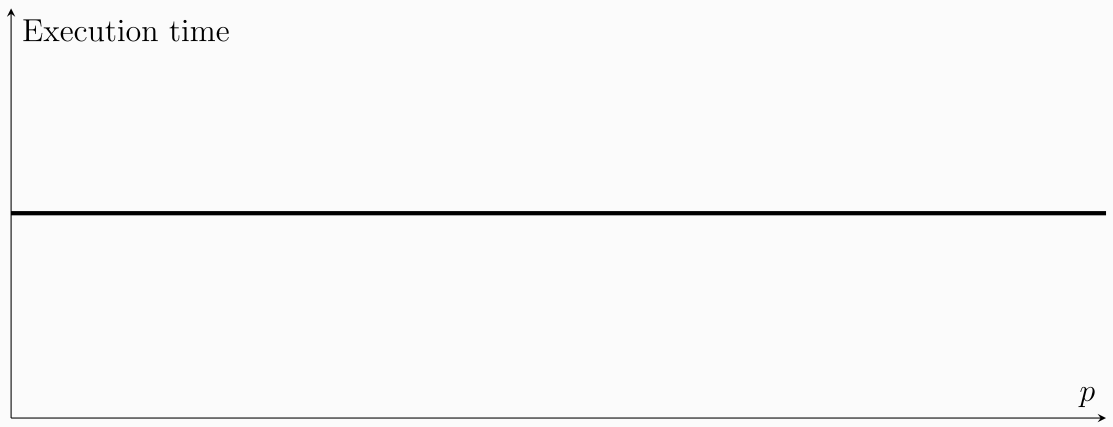
(i) Sketch an approximate plot showing how the execution time of the model varies with p.

(ii) Sketch another approximate plot showing how the model perplexity varies with p.

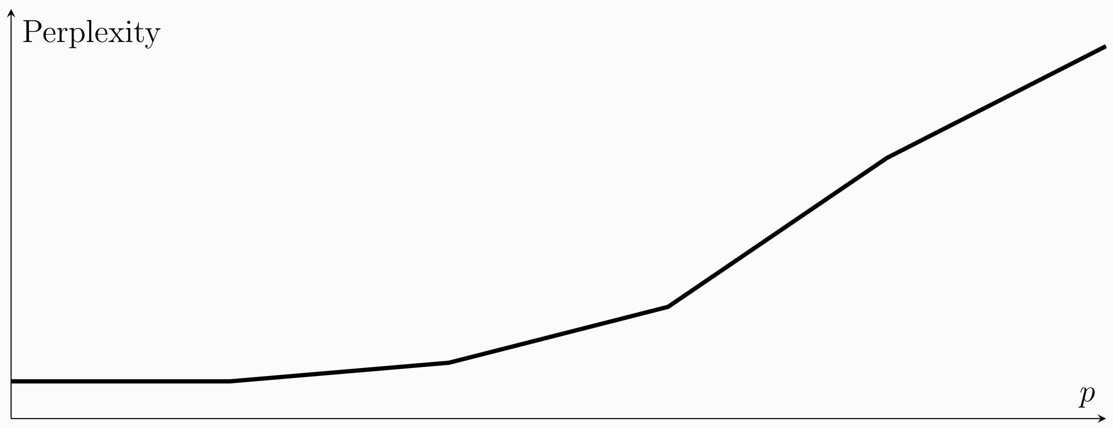
Also provide brief explanation for the trends. *(1 + 1 = 2 marks)*

Answer

(i) Execution time vs pruning p:



(ii) Perplexity vs pruning p:



Explanation: Since pruning is unstructured and only masks weights, runtime remains constant with p. Perplexity remains flat for small pruning but rises steeply after a threshold as too many informative weights are removed.

♣ ♠ ♡ ♢